View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Udinus Repo

Seminar Nasional Teknologi Informasi dan Multimedia 2014 ***ISSN : 2302-3805***
STMIK AMIKOM Yogyakarta, 19 Februari 2014

TRANSLATOR NOTASI ALGORITMIK DENGAN LL(*) PARSING DAN STRING TEMPLATE

Wijanarto[1)], Ajib Susanto[2]

[1), 2)] Teknik Informatika UDINUS Semarang
Jl Nakula 5 -11,Semarang 50131
Email : wijanarto.nagan@yahoo.com[1)], ajibsusanto@gmail.com[2)]

Abstrak

Pemrograman dasar merupakan pondasi utama seseorang atau mahasiswa yang ingin belajar membuat program untuk menyelesaikan suatu masalah tertentu. Kesulitan utama seseorang dalam membuat solusi dalam bentuk bahasa formal merupakan masalah tersendiri, selain pemilihan alat atau aplikasi yang tepat untuk membantunya, bahkan untuk orang dengan latar belakang ilmu komputer. Paper ini mencoba menghasilkan Domain Specific Language (DSL) untuk pengajaran pemrograman dasar dengan grammar LL(*), dalam suatu rancangan aplikasi untuk mempermudah penyelesaian masalah dibidang pengajaran pemrograman dasar berbasis notasi algoritmik. Model notasi algoritmik yang di pilih merupakan model yang sudah pernah diterapkan dan diajarkan di perguruan tinggi. Grammar dihasilkan dengan bantuan ANTLR dan string template, yang di sesuaikan dengan model yang di pilih. Hasil dari penelitian ini berupa Editor Translator Notasi Algoritmik (ETNA), yang diperuntukan bagi mahasiswa di tahun pertama, yang dapat mentranslasikan notasi algoritmik ke bahasa c standar. Alat ini diharapkan membantu seseorang atau mahasiswa untuk dapat mendisain solusi dalam bentuk notasi algoritmik, tanpa memikirkan kerumitan dalam bahasa yang dipakai.

Kata kunci: *Translator, Notasi Algoritmik, Pemrograman, domain specific language.*

1. Pendahuluan

Pemrograman dasar merupakan pondasi utama sesorang atau mahasiswa yang ingin belajar membuat program untuk menyelesaikan suatu masalah tertentu. Sesederhana apapun, masalah yang harus di pecahkan harus dilakukan secara terstruktur dan ilmiah. Dalam dunia ilmu komputer atau teknik informatika langkah-langkah pemecahan masalah atau metode yang logis, terstruktur dan berhingga di sebut sebagai algoritma [1,4]. Seperti diketahui algoritma merupakan metode penyelesaian masalah yang umum dan banyak di lakukan hampir di seluruh bidang ilmu [3], sepert teori graph dalam menentukan lintasan terpendek [7] dan masih banyak lagi. Dalam studi yang pernah dilakukan di Afrika Selatan [2], keberhasilan pembelajaran pemrograman dasar di pengaruhi oleh, (1) lingkungan belajar (alat atau aplikasi) yang mendukung notasi yang sederhana, yang dapat mengkonstruksi notasi umum untuk bahasa pemrograman, (2) penampilan visual dari struktur program harus memungkinkan mahasiswa pemrograman dasar dapat memahami semantik konstruksi program dan (3) lingkungan kerja aplikasi harus melindungi mahasiswa untuk tidak melakukan interpretasi dan pemahaman yang salah. Di lain pihak pemahaman mahasiswa atau orang yang tertarik mempelajari pemrograman sering terkendala oleh bagaimana menggunakan bahasa itu sendiri. Artinya kesulitan utama mempelajari pemrograman di karenakan kesulitan bagaimana memahami semantik dari suatu bahasa pemrograman, seperti di jelaskan dalam [2]. Di Indonesia studi mengenai pembelajaran pemrograman dasar sangat sedikit, apalagi yang menyangkut alat penunjang atau ketepatan penggunaan aplikasinya. Dalam penelitian yang di lakukan Hidayanti [5], lebih menyoroti metode pembelajaran dari aspek pedagogik, di mana capaian mahasiswa dalam belajar pemrograman dasar sangat rendah di karenakan rendahnya partisipasi, keaktifan dalam berdiskusi dan bertanya serta menjawab pertanyaan dalam kuliah. Sedangkan peneliti lain [16], dalam matakuliah sejenis yaitu komputer dasar, menyimpulkan (masih dari aspek pedagogik) bahwa metode belajar berbasis pada masalah dapat meningkatkan pemahaman materi dan prestasi mahasiswa, namun hanya efektif di lakukan dalam satu siklus saja. Dengan demikian diperlukan model yang dapat menyederhanakan struktur dan semantik instruksi, sehingga dapat mempermudah pemahanan serta mengurangi interpretasi yang salah dalam rangka menyelesaikan masalah dalam bidang pemrograman. Paper ini akan mencoba menghasilkan prototype translator notasi algoritmik ke dalam bahasa C standard untuk pengajaran pemrograman dengan grammar

Domain Spesific Language (DLS), sudah muncul sejak lama [22, 23] juga meta programming [21], konsep ini merupakan perbaruan dari teknik konstruksi kompiler yang secara modern [17, 19, 20, 28]. Parsing dan analisa syntax atau lexical (lexer) [13,18, 24, 25, 26, 27] yang merupakan penentu perluasan ekspresi reguler menjadi pokok perhatian dalam membangun translasi dari kode ke bahasa mesin yang di mengerti komputer. Domain Spesific Language (DSL),

merupakan bahasa pemrograman yang di tujukan untuk keperluan masalah dan solusi yang spesifik. Dalam implementasi DSL paradigma MVC banyak digunakan [12, 23].

Batasan kontekstual dan semantik suatu sintak merupakan aspek dari bahasa pemrogram perlu di tentukan, setelah itu baru kita tentukan apakah bahasa tersebut formal atau informal. Dalam praktik, sintak biasanya menggunakan BNF (Backus-Naur Form) atau Extended BNF, karena kemudahan notasinya [18, 19, 20], yang terdiri dari *himpunan berhingga simbol terminal, simbol non terminal, simbol awal dan aturan produksi* $N ::= \alpha \mid \beta$, dimana **N** adalah simbol *non terminal* , **::=** berarti *terdiri dari* serta **α** adalah *string terminal* atau *non terminal* yang mungkin kosong serta simbol | yang berarti *alternatif*, himpunan tadi di sebut sebagai *context-free grammar*, singkatnya *grammar*.

Parsing LL(*) [12] merupakan perbaikan dari LL(*k*) untuk *k*>1, *lookahead* pada LL(*k*) terbatas pada *k* saja, sedangkan dalam LL(*) dapat mengestimasikan berapa kedalaman *lookahead*. Definisi formal dari LL(*), sebagai berikut [13] grammar *G=(N,T,P,S,Π,M),* dimana *N* adalah himpunan non terminal simbol atau rule, T adalah himpunan terminal simbol atau token, *P* adalah himpunan produksi, $S \in N$ merupakan start simbol, *Π* himpunan side effect free predikat semantik, dan *M* adalah himpunan aksi (mutator). Gambar 1 berikut selengkapnya mengenai notasi predikat grammar pada LL(*).

$A \in N$	Nonterminal
$a \in T$	Terminal
$X \in (N \cup T)$	Grammar symbol
$\alpha, \beta, \delta \in X^*$	Sequence of grammar symbols
$u, x, y, w \in T^*$	Sequence of terminals
$w_r \in T^*$	Remaining input terminals
ϵ	Empty string
$\pi \in \Pi$	Predicate in host language
$\mu \in \mathcal{M}$	Action in host language
$\lambda \in (N \cup \Pi \cup \mathcal{M})$	Reduction label
$\vec{\lambda} = \lambda_1..\lambda_n$	Sequence of reduction labels
Production Rules:	
$A \rightarrow \alpha_i$	i^{th} context-free production of A
$A \rightarrow (A'_i) \Rightarrow \alpha_i$	i^{th} production predicated on syntax A'_i
$A \rightarrow \{\pi_i\}? \ \alpha_i$	i^{th} production predicated on semantics
$A \rightarrow \{\mu_i\}$	i^{th} production with mutator

Gambar 1. Notasi Predikat Grammar LL(*)

Produksi adalah di nomori untuk mengekspresikan sebelumnya, sebagai alat mengatasi ambigu. Produksi yang pertama mewakili standar CFG, kedua, menotasikan jembatan *syntatic predicate*, jika simbo A diperluas ke α_i hanya jika input saat ini juga tepat ketemu dengan sintak yang di deskripsikan oleh *A'i*. Ketiga menotasikan *semantic predicate*, simbol A di perluas ke α_i hanya jika π_i memenuhi konstruksi state. Bentuk terakhir menotasikan aksi, yaitu pengaplikasian rule state berdasarkan mutator μ_i. Aturan turunan predikat grammar untuk mendukung *syntatic*, *semantic* predikat dan *mutator* serta referensi aturan yang di sajikan pada gambar 2 berikut :

$$\textit{Prod}\ \frac{A \rightarrow \alpha}{(\mathbb{S}, uA\delta) \Rightarrow (\mathbb{S}, u\alpha\delta)} \qquad \textit{Action}\ \frac{A \rightarrow \{\mu\}}{(\mathbb{S}, uA\delta) \overset{\mu}{\Rightarrow} (\mu(\mathbb{S}), u\delta)}$$

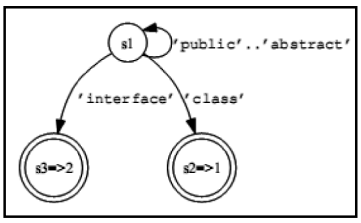
$$\textit{Sem}\ \frac{\pi_i(\mathbb{S}) \quad A \rightarrow \{\pi_i\}?\ \alpha_i}{(\mathbb{S}, uA\delta) \overset{\pi_i}{\Rightarrow} (\mathbb{S}, u\alpha_i\delta)} \qquad \textit{Syn}\ \frac{(\mathbb{S}, A'_i) \Rightarrow^* (\mathbb{S}', w) \quad w \preceq w_r \quad A \rightarrow (A'_i) \Rightarrow \alpha_i}{(\mathbb{S}, uA\delta) \overset{A'_i}{\Rightarrow} (\mathbb{S}, u\alpha_i\delta)}$$

$$\textit{Closure}\ \frac{(\mathbb{S}, \alpha) \overset{\lambda}{\Longrightarrow} (\mathbb{S}, \alpha'), (\mathbb{S}, \alpha') \overset{\vec{\lambda}}{\Longrightarrow}^* (\mathbb{S}, \beta)}{(\mathbb{S}, \alpha) \overset{\lambda\vec{\lambda}}{\Longrightarrow}^* (\mathbb{S}, \beta)}$$

Gambar 2. Predikat grammar dengan aturan turunan terkiri (Leftmost derivation rule)

LL(*) tidak merubah strategi *recursive descent parser*, dia hanya memperluas kemampuan memprediksi keputusan atas predikat pada LL. LL(*) dapat di pakai untuk membangun grammar dan secara otomatis melakukan *left-factoring* untuk mengenerate keputusan yang efisien. LL(*) secara otomatis akan menentukan kedalaman *lookahead*, di bandingkan dengan LL(*k*), dengan *k* lookahead. LL(*) juga mengijinkan *cyclic* DFA seperti pada gambar 3, DFA dengan loop yang dapat memeriksa urutan input *lookahead* yang membedakan alternatif. Perhatikan contoh rule dengan cyclic DFA di bawah ini

```
def : modifier* classDef
    | modifier* interfaceDef
    ;
```



Gambar 3. Cyclic DFA

Cyclic DFA dapat dengan mudah menghindari **modifier** menuju **class** atau **interface**, karena LL(*) akan menghasilkan skipping seperti gambar 4 berikut,

```
void def() {
  int alt=0;
  while (LA(1) in modifier) consume(); // scan past modifiers
  if ( LA(1)==CLASS ) alt=1;           // 'class'?
  else if ( LA(1)==INTERFACE ) alt=2;  // 'interface'?
  switch (alt) {
    case 1 : ...
    case 2 : ...
    default : error;
  }
}
```

Gambar 4. Teknik cyclic DFA pada LL(*)

Dalam kasus ini LL(*) akan melakukan loop sederhana yang di implementasikan pada rule **def** untuk memprediksi DFA. LL(*), juga mampu mengkonstruksi syntatic dan semantic predicate. Semantik mengacu pada sesuatu di belakang syntax atau semua hubungan antar simbol input kepada intepretasi statement. Perhatikan contoh rule dengan elemen matching lebih dari empat kali,

```
data :      BYTE BYTE BYTE BYTE
```

```
|       BYTE BYTE BYTE
|       BYTE BYTE
|       BYTE
;
```

Dengan CFG kita akan sukar menghitung kemungkinan kombinasinya (tanpa sematik predikat), saat kombinasinya melebihi empat, solusinya adalah dengan membuat semantik predikat

```
data: (b+=BYTE)+
{if($b.size()>4)«error»;};
```
atau
```
data: ( b+=BYTE )+ {$b.size()<=4}?;
```

Syntaic predicate sangat berguna pada dua situasi, saat LL(*) tidak dapat menangani grammar dan saat terdapat pecedence diantara dua alternatif yang ambigu.
Di lain pihak grammar merupakan scanner token dengan teknologi LL(*) untuk menangani dan mengenali valid input. Translasi suatu bahasa ke bahasa lain, membutuhan teknik yang di pakai di sini yaitu string template. String Template (ST) [12,14] merupakan engine template dan file template yang di pakai bersama-sama sebagai *controller* untuk melakukan translasi. ST merupakan DSL untuk mengenerate teks terstruktur dari internal struktur data untuk output suatu grammar. ST program dapat di tulis dalam java yang merupakan controller dalam finite state automata. Struktur ST dapat terdiri sebagai berikut pada gambar 5,

```
group groupname;
template1(a1, a2, ..., an) ::= "..."
...
```

Gambar 5. Struktur Group Template

Template berisi kumpulan referensial mutual pada output yang menyediakan pustaka untuk mengkonstruksi output bagi kontroler. Template di kompilasi menjadi instance bertipe string template yang bertindak sebagai prototype instance selanjutnya. Setiap instance template yang berisi suatu tabel pada instance tertentu dengan pasangan nilai dan nama atribut dan sebuah struktur abstract syntax tree (AST) yang di share oleh seluruh instance yang mewakili literal dan ekspresi. AST yang tersedia di pakai untuk mempercepat interpretasi dari template selama program berjalan. Himpunan atribut dalam tabel atribut di batas pada daftar argumen formal a_i. Template merupakan fungsi yang memetakan suatu atribut atau kumpulan atribut ke atribut lainnya dan menspesifikasi melalui pemilihan daftar literal output, t_i, dan ekspresi e_i, dengan demikian fungsi a_i seperti gambar 6 berikut,

$$F(a_1, a_2, ..., a_m) ::= \text{“}t_0 e_0 ... t_i e_i t_{i+1} ... t_{n_t} e_{n_e}\text{”}$$

Gambar 6. Fungsi String Template

Dimana t_i, mungkin string kosong, e_i terbatas pada sintak dan komputasional untuk memperkuat pembagian model-view. Notasi e_i berada dalam tanda kurung **<. .>** yang megelilinginya. Jika tidak terdapat e_i maka template hanya terdiri literal tunggal saja, yaitu t_i, . Operator konkatenasi yang tidak terlihat di aplikasikan pada tiap elemen selanjutnya, evaluasinya di picu dengan konverter method `toString()`[14].

2. Pembahasan

Translator notasi algoritmik di implementasikan dengan ANTLR dan string template dalam bentuk suatu editor dan command line, yang di namakan Editor Translator Notasi Algoritmik (ETNA). Model notasi algoritmik yang di pilih sudah di ajarkan di lingkungan universitas, arsitektur yang sudah di kembangkan juga telah di tentukan [8,15]. Pendekatan yang di pakai dalam implementasi arsitektur ini adalah MVC (Model View Controller) berbasis pada paradigma object oriented yang di tulis dalam java sebagai target aplikasi. Grammar *Algoritmik.g* ditulis sebagai pengenal input yang akan di generate sebagai parser dan lexer dengan teknik LL(*), begitu juga di tulis kode group template *Algoritmik*.stg yang bertindak sebagai kontroler, yang mentranslasikan input token ke bahasa yang dikehendaki (bahasa c), berikut potongan grammar yang sudah di tulis pada gambar 7.

```
grammar Algoritmik;
......
Program
...
: declaration+
  -> program(
     libs={$program::libs},
     globals={$program::global
     functions={$program::functions},
     mainfunctions={$program::mainfunctions}
);
....
COMMENT
:'/*'(options{greedy=false;}:.)*'*/'
{$channel=HIDDEN;}
;
LINE_COMMENT
         :'//'~('\n'|'\r')*'\r'?'\n'
              {$channel=HIDDEN;}
```

Gambar 7. Grammar Algoritmik.g

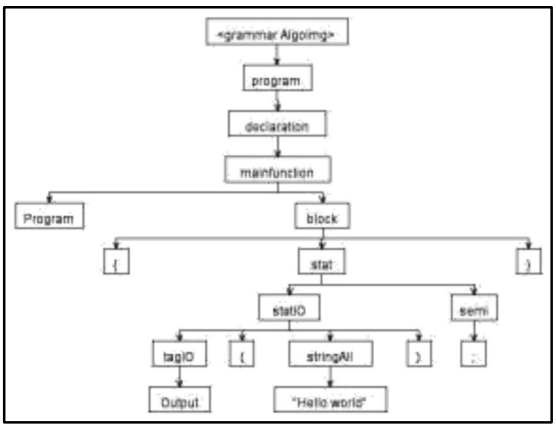
Group template dari grammar di atas di sajikan dalam potongan kode pada gambar 8 berikut

```
group Algoritmik;
program
(libs,globals,functions,mainfunctions)
::=<<
<libs; separator="\n">
<globals; separator="\n">
<functions; separator="\n">
<mainfunctions; separator="\n">
                    >>
...
```

Gambar 8. Template Algoritmik.stg

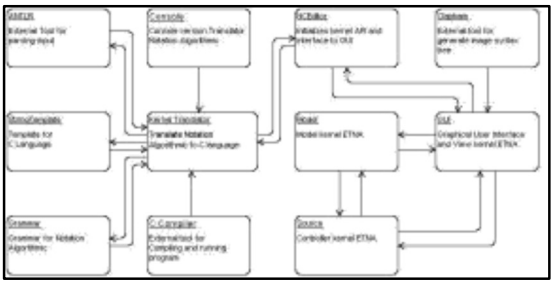
Output abstract syntax tree yang di hasilkan seperti pada gambar 9 berikut, misalkan di berikan input seperti berikut,

```
Program {
Output ("Hello World");
}
```



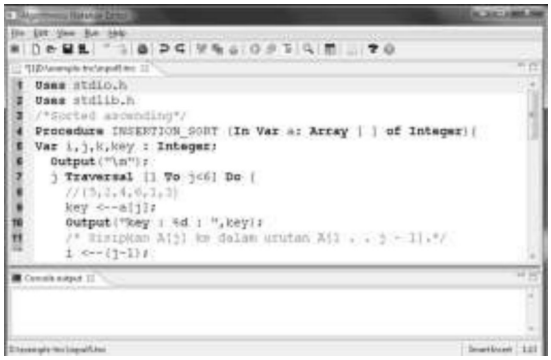
Gambar 9. Abstract Syntax Tree dari Grammar

Pembangunan editor berdasarkan pendekatan MVC dalam java seperti diagram block pada gambar 10 berikut,



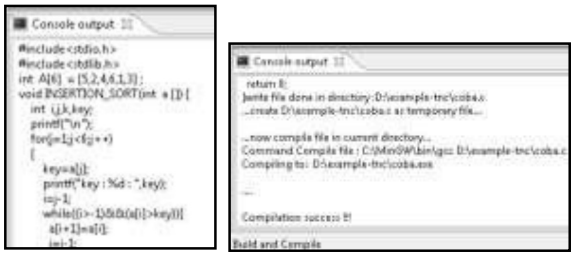
Gambar 10. Block Diagram ETNA

Block Diagram disini di gunakan untuk menjelaskan detail ETNA, yang terdiri dari kumpulan paket dan kelas yang terintegrasi Seperti terlihat, Translator (kernel ETNA), saling berkomunikasi dengan ANTLR, String Template dan grammar, sebagai paket dan kelas yang di pakai kernel. Parser dan Lexer dari grammar yang di hasilkan ANTLR, serta string template yang di tulis khusus untuk bahasa c (disesuaikan kebutuhan), dipakai oleh kernel selama ETNA berjalan. GUI sebagai interface ETNA dan user memakai kernel saat diperlukan. Console merupakan translator dalam versi command line yang memakai kernel serta kompiler c sebagai tool luar untuk menghasilkan file eksekusi juga di pakai oleh kernel. Interaksi kernel dan GUI (ETNA), melalui NCEditor, saat aplikasi dimulai kernel akan di inisialisasikan oleh NCEditor bersama-sama GUI sekaligus sebagai viewer, model dokumen serta source controller, sebagai implementasi model MVC. Sementara tool dari luar Clapham di pakai menggenerate image syntax tree yang saat ini di pakai untuk membantu user memahami notasi agoritmik (ke depan akan di manfaatkan untuk error trace secara visual). Seperti terlihat Kernel dan GUI tidak berkomunikasi secara langsung, tapi melalui Model dan Source yang di hubungkan oleh NCEditor untuk berkomunikasi dengan Kernel, dimana parser, lexer serta string template juga melalui kernel dan NCEditor untuk berkomunikasi dengan GUI sebagai interface user. Setelah di implementasikan dengan java ETNA berhasil di kembangkan dan sukses melewati beberapa uji fungsional, berikut tampilan ETNA pada gambar 10.



Gambar 10. Tampilan ETNA dengan file notasi algoritmik aktif

Sementara hasil translasi oleh ETNA dapat di lihat pada gambar 11 a, b dan c berikut ini,



(a) (b)

(c)

Gambar 11 (a). Hasil translasi, (b) Hasil kompilasi, (c). Hasil eksekusi

Masing-masing gambar (a) merupakan hasil translasi dari notasi algoritmik, sedang (b) saat ETNA melakukan kompilasi file hasil translasi menjadi executable file dan (c) saat ETNA menjalankan file hasil kompilasi dan sukses.

3. Kesimpulan

Dari hasil implementasi translator notasi algoritmik dengan teknik parsing LL(*) dan string template dalam

suatu aplikasi sistem translator notasi algoritmik, penulis dapat menyimpulkan sementara bahwa dari model translator yang di pilih dapat membantu pemakai (mahasiswa) untuk memecahkan masalah pemrograman dasar dalam bentuk notasi, tanpa perlu memahami bahasa yang di pakai. pembelajaran pemrograman dasar. *ETNA* berhasil dibangun dan sudah memiliki fitur yang cukup untuk membantu dalam menulis notasi (*code completion, syntax highlight, error correction*). Kemampuan mentranslasikan, mengkompilasi dan dapat menjalankan program *on the fly* juga lebih tepat dapat memberi informasi mengenai kegagalan atau keberhasilan solusi yang di tulis dengan notasi algoritmik. Kedepan *ETNA* ini perlu di lengkapi representasi *visual syntax tree* mengenai notasi algoritmik, yang muncul ketika di temukan kesalahan kode dan memunculkan grafik notasi syntax tree tepat di mana letak kesalahan di temukan serta bagaimana notasi yang benar, sehingga pengguna dapat memahami *syntax* yang benar dan segera membetulkan kesalahannya

Daftar Pustaka

[1] Blass, Andreas; Gurevich, Yuri., 2003, Algorithms: A Quest for Absolute Definitions, Bulletin of European Association for Theoretical Computer Science.

[2] Chairmain Cilliers, Andre Calitz, Jean Greyling, 2005, *The Application of The Cognitive Dimension Framework for Notations as an Instrument for the Usability analysis of an Introductory Programming tool*, Alternation Journal, 12.1b, p 543-576 ISSN 1023-1757.

[3] Chen Shyi-Ming, Lin Chung-Hui, Chen Shi-Jay, 2005, *Multiple DNA Sequence Alignment Based on Genetic Algorithms and Divide-and-Conquer Techniques*, International Journal of Applied Science and Engineering. 3, 2: 89-100.

[4] David Harel, Yishai A. Feldman, 2004 , *Algorithmics: the spirit of computing, Edition 3*, Pearson Education, ISBN 0-321784-0.

[5] Hindayati Mustafidah, 2007, *Prestasi Belajar Mahasiswa dalam Mata Kuliah Pemrograman Dasar Melalui Pembelajaran Kooperatif Model Jigsaw*, Paedagogia, Agustus jilid 10 No 2, hal. 126 – 131.

[6] Ian Somerville, 2011, *Software engineering, 9th edition*, Pearson Education, Addison-Wesly, Boston, Massachusetts.

[7] Kruskal J. B, Jr., 1956, *On the shortest spanning subtree of a graph and the traveling salesman problem*. Proceedings of the American Mathematical Society, 7, pp. 48-50.

[8] Liem, Inggriani, 2007, *Draft Diktat Dasar Pemrograman (Bagian Prosedural)*, ITB , Bandung, unpublished.

[9] Reenskaug, Trygve M.H., 1979, *MODELS - VIEWS - CONTROLLERS.* , XEROX PARC.

[10] Reenskaug, Trygve M.H., 1979, *THING-MODEL-VIEW-EDITOR an Example from a planningsystem.* , Xerox PARC technical note May 1979.

[11] Stanchfield, Scott. *Applying MVC in VisualAge for Java. JavaDude*. [Online] 1996 - 2009. diakses: 10-10-2012. http://javadude.com/articles/vaddmvc2/mvc2.html.

[12] Parr, Terrence, 2006, A Functional Language For Generating Structured Text, di akses 10-10-2013, 2006, http://www. cs. usfca. edu/parrt/papers/ST. pdf

[13] Parr, Terrence, Fischer, Kathleen S, 2011, LL(*) : The Foundation of the ANTLR Parser Generator, PLDI '11, Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, ACM New York, NY USA, ISBN: 978-1-4503-0663-8

[14] Parr, Terrance, Fischer, Kathleen S, 2004, Enforcing Strict Model-View Separation in Template Engines, New York, New York, USA. ACM 1-58113-844-X/04/0005

[15] Wijanarto, Achmad Wahid Kurniawan, 2012, *Model Translator Algoritmik ke Bahasa C*, Prosiding Kommit, Komputer dan Sistem Intelijen, Vol 7, 464-472 ISSN 2302-3740.

[16] Yuwono Indro Hatmojo, Sigit Yatmono, 2009, *Peningkatan Prestasi Mata Kuliah Komputer Dasar Mahasiswa D3 Teknik Elektro FT UNY Menggunakan Metode Belajar Berbasis Masalah*, Jurnal edukasi@Elektro Vol. 5, No.1, Maret, hal. 67 – 78.

[17] Alfred V Aho, Monica S Lam, Ravi Sethi , Jeffrey D Ullman. 2007. *Compilers : principles, techniques, and tools Second Edition.* New York : Pearson Education Addison Wesly, 2007.

[18] Alverd V Aho, Jeffery D Ullman. 1973. *The Theory of Parsing, Translation and Compiling.* New York : Prentice Hall Englewood Cliffs, 1973. 0-13-914564-8 .

[19] Andrew W Appel, Maia Ginsburg. 1998. *Modern Compiler Implementation In C.* New York : CAMBRIDGE UNIVERSITY PRESS, 1998.

[20] David A Watt, Deryck F Brown. 200. *Programming Language Processors in Java, Compiler and Intepreter.* New York : Pearson Education, Addison Wesly, 200.

[21] Dimitriev, Sergey. 2004. Language Orintation Programming : The Next Programming Paradigm. [November] s.l. : JetBrain, 2004.

[22] Fowler, Martin. 1999. *Analysis Patterns : Reuseable Object Models.* New York : Addison Wesley, 1999.

[23] —. 2010. *Domain Specific Languages* . New York : Addison-Wesley Professional , 2010. ISBN-10: 0-321-71294-3 .

[24] Gijzel, Bas van. 2009. *Comparing Parser Construction Techniques.* s.l. : University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, 2009.

[25] Hanson, Ralph E. Griswold and David R. 1980. *An Alternative to the Use of Patterns in String Processing.* Vol. 2 Issue 2 Pages 153-172 : ACM Transactions on Programming Languages and Systems, 1980.

[26] John R. Levine, Tony Mason, Doug Brown. 1992. *lex & yacc* . California : O'Reilly & Associates, Inc. 103 Morris Street, Suite A Sebastopol, CA 95472 , 1992. ISBN: 1-56592-000-7 .

[27] Keith D Cooper, Linda Torczon. 2003. *Engineering a Compiler, Second Edition.* San Francisco : ISBN: 1-56592-000-7 , 2003. ISBN-13: 978-1558606982 .

[28] P. Rechenberg, H. Mocchenbock. 1989. *A Compiler Generator For Microcomputers.* London : Prentice Hall International UK, 1989. ISBN : 0-13-155060-8.

Biodata Penulis

Wijanarto, Memperoleh gelar Magister Komputer (M.Kom) Program Pasca Sarjana Magister Ilmu Komputer Universitas Gajah Mada Yogyakarta, lulus tahun 2006. Saat ini menjadi Dosen di UDINUS Semarang.

Ajib Susanto, memperoleh gelar Sarjana Komputer (S.Kom), Jurusan Teknik Informatika UDINUS Semarang, lulus tahun 2004. Memperoleh gelar Magister Komputer (M.Kom) Program Pasca Sarjana Magister Teknik Informatika Universitas Dian Nuswantoro Semarang, lulus tahun 2008.Saat ini menjadi Dosen di UDINUS Semarang.